TDTS06

Lab 3 report

Mayeul Gasc (mayga330@student.liu.se)
Pierre Marsaa (piema262@student.liu.se)

TCP Basics

1) The first packet is the packet n° 4 because it's the one just after the three-way handshake. And the last one is the packet n° 203 because it says 200 OK.

2) &
3) On the first line (representing the first captured packet), we can read:

1 192.168.1.102 128.119.245.12 TCP 1161 → 80

where: Pkt n° IP Source IP Dest Protocol PortSource → PortDest

So, we have :

IP source: 192.168.1.102 Port source: 1161
IP destination: 128.119.245.12 Port destination: 80

4) The sequence number in the TCP SYN packet is 232129012, the SYN segment is identified by the SYN flag set.

5) The SYN sequence number in the SYNACK packet is 883061785. The ACK sequence number is 232129013, which is the sequence number in the TCP SYN plus one. The SYNACK packet is identified as such by the flags SYN and ACK set.

6) The sequence number of the TCP segment containing the POST request is 232129578. (Packet n° 4 in the trace.)

7) By analysing the trace, we have:

Packet n° in the trace	**Sequence number**	**Time sent** (since first frame*)	**ACK received time**	**RTT**
4	232129013	0.026477	0.053937	0,02746
5	232129578	0.041737	0.077294	0,035557
7	232131038	0.054026	0.124085	0,070059
8	232132498	0.054690	0.169118	0,114428
10	232133958	0.077405	0.217299	0,139894
11	232135418	0.078157	0.267802	0,189645

* knowing that the first frame in the TCP stream occurs at: 1093095860.5703810s (epoch time).

According to the course book, for each packet the Estimated RTT is given by:

$$EstimatedRTT = (1 - a) * EstimatedRTT + a * SampleRTT$$

where a = 0.125 (RFC 6298). So, we have:

RTT	Estimated RTT
0,02746	0,02746
0,035557	0,028472
0,070059	0,033670
0,114428	0,043765
0,139894	0,055781
0,189645	0,072514

8) The length of the first TCP ((POST request) is 565 bytes and the length of all the others is 1460 bytes.

9) We can read the windows size in the SYN/ACK packet during the hand-shake: 5840. The size of the window gradually increases and never throttles the sender.

10) All the sequence numbers are different, so it can be inferred that there was no packet loss.

11) Typically an ACK acknowledges 1460 bytes. This size can be found in the ACK sequence number, which seems to be calculated as follows: *seqNum = precSeqNum + size*.
We can observe some irregularity of this rule, e.g. between ACK 102045 (packet 132) and ACK 104965 (packet 133) where the difference is 2920 = 2*1460. We therefore observe here an ACK packet which acknowledges all the packets which have not yet been ACK.

12) By looking at the “follow TCP flux” windows, we see a content_length field of 163411 bytes and we know that the transmission lasted 5.45583-0.04173 = 5.41401 seconds. We deduce a throughput of 30182 bytes/s ⇔ 30,182 kbytes/s

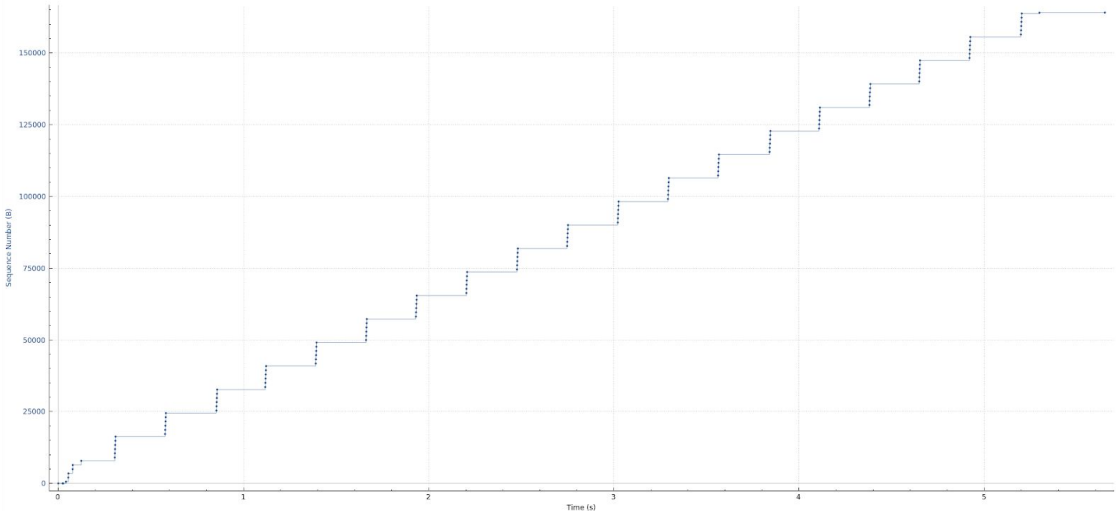
(Task A) A TCP connection begins with a handshake which allows the sender and receiver to share their sequence numbers. These numbers will make it possible, via the sending of an ACK packet, to confirm the number of bytes received. In this way, it is ensured that all the data reaches the receiver.

The TCP protocol must therefore ensure that no packets are lost. In this context, the estimated RTT plays an important role since it makes it possible to estimate from when it is possible to estimate that a packet is lost, in order to send it again. This estimate can be calculated simply as we saw in question 7. Moreover, its value is constantly adjusted according to the fluctuations of the connection.

Note: In the event that we want to optimize the use of the throughput as much as possible, the RTT is also important since it makes it possible to estimate the number of packets "in flight" and to adjust the send window according to that in order to be at the limit of saturation.

Finally, it is a pity that the trace that we must observe does not present more problems, this would have allowed us to better observe and understand what happens when a packet is lost.

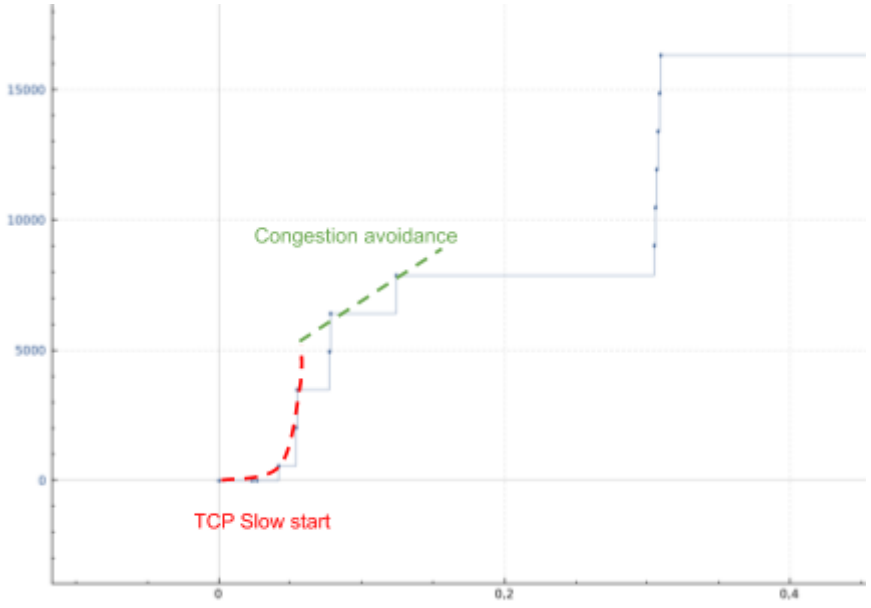
TCP Congestion Control in Action



A screenshot of the time sequence Steven graph.

(Task B)

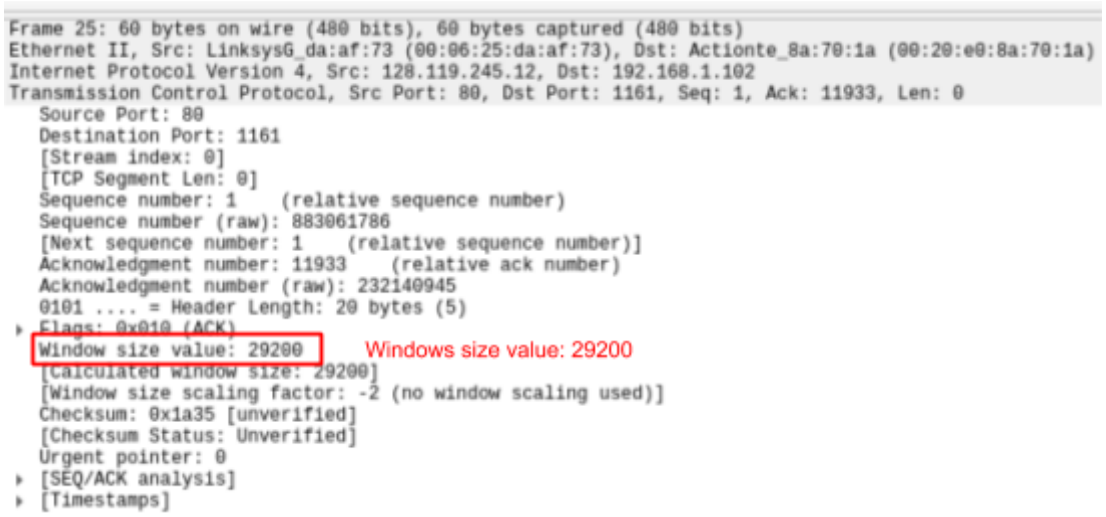
13) By zooming on the graph we can clearly see the TCP slow start:



Screenshot zoomed on edge (start)

If the TCP stream behaves as in the text, we should see this pattern repeat itself several times. Here, on the contrary, we observe "falls" of TCP packets. This may be due to an application bandwidth limit (i.e. the app does not use all the bandwidth, only a part).

14) The (i) *cwnd* size is fixed by the sender and limits the number of bytes that can be sent before receiving the next ACK. The (ii) *rwnd* size must be greater or equal than the *cwnd* size since it is the limit of bytes that can be received. In practice these windows may be equal if the transmission is the application can use all the bandwidth. If these windows allow N bytes and if the transmission is efficient, the (iii) *number of unacknowledged bytes* is N. Eventually, the (iv) the *effective window at the sender* must be smaller or equal to the (i) *cwnd* and so the (ii) *rwnd*.



The receiver windows size can be found in the ACK packets.

15) This value is internal to the sender and we cannot find its value in the trace. However we can estimate it since this value is smaller than the *rwnd* size value. More, as we saw before, this TCP flow does not use all the bandwidth (no congestion avoidance), we can guess that the *cwnd* can be much smaller (if this is the bandwidth limiting factor), but we can not know the size. The only known value is the *rwnd* size.

A Short Study of TCP Fairness

16) From the information in the table, one can simply calculate the throughput of each connections:

N°	Total bytes	Duration	RTT	Bytes/s (rounded)	Bits/s (rounded)
1	165095720	521	12	316882	2535059
2	165842766	521	12	318316	2546529
3	165458792	514	12	321904	2575234
4	163235772	512	12	318819	2550558

We observe that the connection is very fair because the four connections have the same speed. We can calculate the bandwidth by cumulating the speeds (assuming these are the only significant connections):

bandwidth = 2535059 + 2546529 + 2575234 + 2550558 = 10,207,380 bits/s ≅ 10.2 Mbits/s

17) In the same way than before:

N°	Total bytes	Duration	RTT	Bits/s (rounded)	Bits/s (readable)
1	261319130	90	13	23228367	23.2 Mbits/s
2	175995832	90	35	15644073	15.6 Mbits/s
3	151894552	90	68	13501737	13.5 Mbits/s

4	140388568	90	73	12478983	12.5 Mbits/s
5	108610702	90	49	9654284	9.6 Mbits/s
6	70644690	90	33	6279528	6.3 Mbits/s
7	65744938	90	135	5843994	5.8 Mbits/s
8	43212876	90	326	3841144	3.8 Mbits/s
9	39222524	90	322	3486446	3.5 Mbits/s
			Total:	93958561	94.0 Mbits/s

This time, the connection sharing is less fair, the connections having a higher RTT are slower. This can be explained by the equation given at the beginning of the chapter, the greater the RTT, the lower the throughput:

$$Average\ throughput\ of\ a\ connection = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

18) Idem than previously:

N°	Total bytes	Duration	RTT	Bits/s (rounded)
1	108851134	58	40	15013949
2	90435681	58	36	12473887
3	57971584	53	100	8750427
4	32000012	29	68	8827589
5	32557334	35	31	7441676
6	27199361	31	33	7019189
7	26329578	31	122	6794729
8	38834490	56	146	5547784
9	23571761	35	74	5387831
10	36252962	55	66	5273158
			Total:	82530223

As in the previous question, the connections will have a throughput that is inversely proportional to the RTT. However, BitTorent being a peer to peer service, we can assume that it will open several connections with many pairs to download the data. So a single application will open several connections and therefore make the sharing of the connection less fair.